NRL Report 8734
August 23, 1983

Interface Specifications for the SCR (A-7E) Application Data Types Module

Paul C. Clements

Stuart R. Faulk

David L. Parnas

Naval Research Laboratory
Washington, D.C. 20375

Abstract

This document describes the programmer interface to a set of avionics-oriented abstract data types implemented in software. The Application Data Types module is part of NRL’s Software Cost Reduction (SCR) project, to demonstrate the feasibility of applying advanced software engineering techniques to complex real-time systems in order to simplify maintenance. The Application Data Types module allows operations on data independent of its representation. In the case of numeric abstract types, which represent physical quantities such as speed or distance, arithmetic operations may be performed independent of the units of physical measure. This allows the rest of the application software to remain unchanged even when representation decisions about this data change.

This report contains the abstract interface specifications for all of the facilities provided to users by this module. It serves as development and maintenance documentation for the SCR software design, and is also intended as a model for other people interested in applying the abstract interface approach on other software projects.

Keywords

abstract data types
abstract interfaces
avionics software
information hiding
modular decomposition
modules
real-time systems
software engineering
software maintenance
software specifications

Introduction

1. Purpose

This document specifies the abstract interfaces for a software module that forms part of the Operational Flight Program for the Navy's A-7E aircraft. As the demonstration vehicle for the Naval Research Laboratory's Software Cost Reduction (SCR) project, the program is being designed and implemented in accordance with modern software engineering principles such as information-hiding, cooperating sequential processes, and abstract data typing.

The basic computing environment for the application software is provided by the Extended Computer module, specified in reference [EC]. The Extended Computer interface provides all the facilities necessary for handling data, as well as i/o, sequence, and process control.

2. Overview

The Application Data Types (ADT) module provides facilities for handling those data types which (a) are useful to the application at hand (in this case, an embedded real-time avionics program), and (b) can be implemented independently of the host computer; i.e., are not provided by the Extended Computer Module.

Although the type classes provided by the module are fixed, operations are available to specify specific types within a type class, and to create and manipulate data objects of each type. The module comprises two submodules:

(1) Numeric Type Classes. This submodule provides those numeric data type classes that are considered to be generally useful to avionics applications; these type classes are used to represent physical quantities: acceleration (both scalar and vector), acceleration rate, angle, angular rate, density, displacement, distance, orientation, orientation rate, pressure, speed, and velocity. "Power types" (i.e., types raised to integer powers such as 1/distance, speed^2, 1/pressure^2, etc.) are also provided. Facilities are provided that allow manipulation of such quantities without regard to the units of measurement.

(2) State Transition Event (STE) Type Class. This submodule allows programs to create and operate on data types described as finite state machines. The domain of an STE variable is a set of states. The changing of a variable's state is an event that can be signalled and awaited.

3. Conventions of this document

The rules and conventions enumerated in [SO] and [ACONV] apply.

CHAPTER 1

AT.NUM: NUMERIC TYPES AND OPERATIONS

1. Introduction

1.1. Overview

The type classes provided by this submodule give a means to represent and operate on physical quantities (such as a speed or an angle) without stating or assuming particular units of measurement. Conversion programs are provided to convert the entity to its real equivalent, given the unit of measurement desired.

The notion of type classes and specific types is explained in EC.DATA.1.2. The types provided by this module, as well as the available units of measurement for each, are listed in Table AT.NUM.a.

Specific types in this module are characterized only by the range and resolution of all entities of that type. Specific types are created via declarations. Each variable and constant must belong to a specific type.

1.2. Literals

1.2.1. ADT scalar literals

A value for a scalar type provided by this module can be specified by using a real literal or ascon and one of the conversion programs that convert real values to numeric scalar abstract types. The syntax is:

< program-name real >

OR (for angle literals from degree/minute/second reals)

< +ANGLE_R_DMS+ real real real >

OR (for angle literals from sine/cosine reals)

< +ANGLE_R_SINCOS+ real real integer>

"Real" stands for a real-literal or a previously-declared real ascon. "Integer" stands for an integer literal or a previousy declared real ascon whose value is an integer. The syntax for real-literal is given in EC.DATA.3. The value thus specified is that which would be returned by the program were the real(s) given as the input parameter(s). For +ANGLE_R_SINCOS+ the integer is optional; if omitted, zero will be used.

1.2.2. ADT vector literals

A value for a vector type can be specified by naming a program that produces the desired type from constituent scalars, which are specified as shown above. The syntax is:

< program-name scalar-literal scalar-literal ... scalar-literal >

1.2.3. Real values from ADT ascons

This module provides a mechanism for producing system-generation-time values of type *real* from literals and ascons of ADT scalar typeclasses. The syntax is:

< program-name ADT-scalar >

where "ADT-scalar" is an ascon of an ADT scalar typeclass, and "program-name" is an AT.NUM program that converts an ADT scalar into a real. The value thus specified is the value that would be returned by the program were the ADT ascon given as the input parameter.

Table AT.NUM.a: Type classes and associated units

Type class†	Unit	Unit meanings
accel	fpss g	feet per second per second normal gravity acceleration
accel-rate	fpsc	feet per second per second per second
accel-vec	See note 1	
angle See note 2	deg min sec cir rad	degrees minutes seconds circles radians
angrate	deghour radsec	degrees per hour radians per second
density	slcuft	slugs per cubic foot
displacement	See note 3	
distance	ft nmi	feet nautical miles
orientation	See note 4	
orient_rate	See note 5	
pressure	inhg	inches of mercury
speed	fps fpm knot	feet per second feet per minute nautical miles per hour
velocity	See note 6	
!!basic scalar type!!^{n} See note 7	!!basic scalar unit!!^{n} See note 7	

† timeint is a type class provided by [EC], although its !!power type!!s are provided by this module. Its available units are ms (milliseconds), sec (seconds), min (minutes), and hour (hours).

Notes:

[1] accel-vec is a three-dimensional vector that may be converted into components of type accel.

[2] Angles may also be converted to a (sine,cosine,circle) triple.

[3] displacement is a three-dimensional vector that may be converted into components of type distance.

[4] orientation is a data type for defining the relationship between two sets of coordinate systems; it may be considered a nine-dimensional vector of reals, or a three-dimensional vector of angles.

[5] orient_rate is a time rate-of-change of orientation; it is a three-dimensional vector that may be converted into components of type angrate.

[6] velocity is a three-dimensional vector that may be converted into components of type speed.

[7] values of *n* are listed in the definition of !!power type!!.

2. Interface Overview

2.1. Declaration of Specific Types

The form is that of the ++DCL_TYPE++ program described in EC.DATA.2.1, with

p1 as described there;
p2 a typeclass as described in AT.NUM.3;
p3 an attribute as described in AT.NUM.3;
p4 a version as described in AT.NUM.3.

2.2. Data Declarations

The form to declare type classes, variables, constants, or arrays is as given in EC.DATA.2.2.

2.3. Operand forms

Operand forms in EC.DATA.2.4 apply. In addition, these two forms apply to operands of the angle typeclass:

keyword ::= MOD1
info ::=

The *parameter* must refer to an entity of the angle typeclass; call it *x*. When used as an !!EC.source!!, the value used in the operation will be the value of *x* modulo* 360°; the value of *x* will not be changed. When used as an !!EC.destination!!, the value stored into *x* will be the result of the operation taken modulo 360°.

keyword ::= MOD2
info ::=

The *parameter* must refer to an entity of the angle typeclass; call it *y*. When used as an !!EC.source!!, the value used in the operation will be *y* modulo 360° - k×360°, where k=0 if *y* modulo 360° ≤180° and k=1 otherwise. When used as an !!EC.destination!!, the value stored into *y* will be the result of the operation taken modulo 360°, after which k×360° will be subtracted, where k is defined as before.

2.4. Transfer Operations

The form is that of the +SET+ and ++SET++ programs described in EC.DATA.2.5; p1 and p2 must both belong to the same type class, or be pointers of the same specific type.

2.5. Numeric Operations

2.5.1. Numeric Comparison Operations

All programs listed in EC.DATA.2.6.1 apply.

——— *Parameters* ———

Parameters p1, p2, and p4 must all belong to the same type class for !!scalar type!!s. For vectors, the following combinations are legal:

p1,p2 accel-vec, p4 accel;
p1,p2 displacement, p4 distance;
p1,p2 orientation, p4 orientation;
p1,p2 orient_rate, p4 orientation;
p1,p2 velocity, p4 speed;

No comparison programs except +EQ+ and +NEQ+ are defined for orientations and orient_rates.

——— *Effects* ———

For operands of scalar type classes, the effects in EC.DATA.2.6.1 apply. For operands of vector type classes, the following program effects apply.

+EQ+ p3 = +ABSV+(+ABSV+(p1) - +ABSV+(p2)) ≤ !!user threshold!! for vectors not of type

* Definition of modulo: Let δ be the greatest integer multiple of β such that δ ≤ α. Then α modulo β ≡ α - δ.

orientation or orient_rate.

+NEQ+	p3 = +ABSV+(+ABSV+(p1) - +ABSV+(p2)) > !!user threshold!! for vectors not of type orientation or orient_rate.
+GT+	p3 = +GT+(+ABSV+(p1), +ABSV+(p2), , !!user threshold!!)
+GEQ+	p3 = +GEQ+(+ABSV+(p1), +ABSV+(p2), , !!user threshold!!)
+LT+	p3 = +LT+(+ABSV+(p1), +ABSV+(p2), , !!user threshold!!)
+LEQ+	p3 = +LEQ+(+ABSV+(p1), +ABSV+(p2), , !!user threshold!!)

2.5.2. Numeric Calculations

All programs in section EC.DATA.2.6.2 apply.

Parameters

+ADD+
+SUB+ All operands belong to the same AT-provided type class.

+ABSV+ All operands belong to the same AT !!scalar type!! class, or

p1 accel-vec, p2 accel; or
p1 displacement, p2 distance; or
p1 velocity, p2 speed.

This program is not defined for orientations or orient_rates.

+COMPLE+ All operands belong to the same AT-provided type class, except orientation or orient_rate.

+DIV+
+MUL+ This module provides multiplication and division of the type combinations in the following table. Any other combination that can be derived using simple algebraic rules (e.g., raising both sides of an equation to the same power, or dividing both sides of an equation by the same thing, or the rules of multiplicative commutativity and associativity) is also allowed, as long as all the types in the resulting equation are among those provided by this module, or real, or timeint.

An exponent of a type is the !!power type index!! of that type. For the purposes of the table, *type*^{0} is considered to be synonymous with *real*.

type^n	×	*type*^{m}	= *type*$^{n+m}$, $n \neq 0$
timeint	×	accel-rate	= accel
timeint	×	accel	= speed
timeint	×	speed	= distance
timeint	×	angrate	= angle
timeint	×	velocity	= displacement
timeint	×	accel-vec	= velocity

+SIGN+ All operands belong to the same AT !!scalar type!! class, or

p2 accel-vec, p1 accel; or
p2 displacement, p1 distance; or
p2 velocity, p1 speed.

This program is not defined for orientations or orient_rates.

Effects

+ABSV+	p2 = magnitude(p1), using the Euclidean norm (see, for example, [NCMA], page 90).
+COMPLE+	p2 = +MUL+(p1, -1).
+DIV+	For dividing a displacement (velocity) by a timeint: p4 = a velocity (accel-vec) vector with the same direction as the displacement, whose magnitude is equal to the magnitude of the displacement (velocity) divided by the timeint.
+MUL+	For multiplying a velocity (accel-vec) by a timeint: p3 = a displacment (velocity) vector with the same direction as the velocity (accel-vec), whose magnitude is equal to the timeint multiplied by the magnitude of the velocity (accel-vec).
+SIGN+	p3 = sign(p1) $\times$ +ABSV+(p2), where sign(0) is defined to be zero.

Program effects for all other cases are defined in EC.DATA.2.6.2.

2.5.3. Operations Converting from AT Scalar Types to Reals

The following specification defines a set of programs; each is defined by replacing *st* with the name of each !!scalar type!! and replacing *bsu* with each !!basic scalar unit!! that applies to the !!basic scalar type!! from which the !!scalar type!! is derived. For example: +R_SPEED_KNOT+, or +R_PRESSURE_INV_INHG+.

Program	*Parameters*	*Description*	*Undesired events*
			%range exceeded%
+R_*st_bsu*+	p1: !!scalar type!!; I	source	
	p2: real; O	destination	
+R_ANGLE_COS+			
+R_ANGLE_SIN+			
+R_ANGLE_TAN+			
+R_ANGLE_COT+	p1: angle; I	source	
	p2: real; O	destination	
+R_ANGLE_DMS+	p1: angle; I	source	
	p2: integer; O	degrees	
	p3: integer; O	minutes	
	p4: real; O	seconds	

Effects

Each program provides a real value in p2 giving the physical quantity of p1 in the !!scalar unit!!s corresponding to the !!scalar type!! in the name of the program. The unit abbreviations are defined in Table AT.NUM.a. Additionally:

+R_ANGLE_COS+	$-1 \leq p2 \leq 1$
+R_ANGLE_SIN+	$-1 \leq p2 \leq 1$
+R_ANGLE_TAN+	p2 = tangent(p1)
+R_ANGLE_COT+	p2 = cotangent(p1)
+R_ANGLE_DMS+	$0 \leq p3 < 60$; $0 \leq p4 < 60$. p2-p4 are set so that p2 degrees + p3 minutes + p4 seconds are equal to the angle given by p1.

2.5.4. Operations Converting from Reals to AT Scalar Types

The following specification defines a set of programs; each is defined by replacing *st* with the name of each !!scalar type!! and replacing *bsu* with each !!basic scalar unit!! that applies to the !!basic scalar type!! from which the !!scalar type!! is derived. For example: +SPEED_R_KNOT+, or +PRESSURE_INV_R_INHG+.

Program	*Parameters*	*Description*	*Undesired events*
			%range exceeded%
+*st*_R_*bsu*+	p1: real; I	source	
	p2: !!scalar type!!; O	destination	
			%illegal sin or cos%
			%range exceeded%
+ANGLE_R_SIN+			
+ANGLE_R_COS+	p1: real; I	source	
	p2: angle; O	destination	
+ANGLE_R_SINCOS+	p1: real; I	sine of angle	
	p2: real; I	cosine of angle	
	p3: angle; O	destination	
	p4: integer; I_OPT	circle nbr	
			%range exceeded%
+ANGLE_R_TAN+	p1: real; I	tangent of angle	
	p2: angle; O	destination	
+ANGLE_R_TAN2+	p1: real; I	sine of angle multiplied by a constant	
	p2: real; I	cosine of angle multiplied by the same constant	
	p3: angle; O	destination	
+ANGLE_R_COT+	p1: real; I	cotangent of angle	
	p2: angle; O	destination	
+ANGLE_R_DMS+	p1: real; I	degrees	
	p2: real; I	minutes	
	p3: real; I	seconds	
	p4: angle; O	destination	

Effects

+ANGLE_R_COS+	p2 = arc cos p1; $0° \le p2 \le 180°$.
+ANGLE_R_DMS+	Produces in p4 the angle equal to p1 degrees + p2 minutes + p3 seconds.
+ANGLE_R_SIN+	p2 = arc sin p1; $-90° \le p2 \le +90°$.
+ANGLE_R_SINCOS+	p3 = (arc sin p1) + p4×360° and p3 = (arc cos p2) + p4×360°. If p4 is omitted, its value is taken to be zero.
+ANGLE_R_TAN+	p2 = arc tan p1; $-90° \le p2 \le +90°$.
+ANGLE_R_TAN2+	p2 = arc tan (p1/p2); $-180° \le p2 < +180°$.

+ANGLE_R_COT+ p2 = arc cotan p1; -90° ≤ p2 ≤ +90°.

All other programs produce an AT-type value in p2 equivalent to p1, assuming that p1 represents a physical quantity in the !!scalar unit!!s corresponding to the !!scalar type!! in the program name. The unit abbreviations are defined in Table AT.NUM.a.

2.5.5. Operations Converting from Vector Types to Scalars

In the following *i*={1,2,3} and *j*={1,2,3}.

Program	*Parameters*	*Description*	*Undesired events*
			%range exceeded%
+ANGLE_ORIENT_*i*+	p1: orientation; I	!+orientation diff+!	
	p2: angle; O	!+euler angle+! #*i*	
+ANGRATE_ORIENT_RATE_*i*+			
	p1: orient_rate; I	!+orient rate diff+!	
	p2: angrate; O	change rate of !+euler angle+! #*i*	
+CYL_VECTOR+	p1: _____; I	!+vector+!	
	p2: _____; O_OPT	!+radius+!	
	p3: angle; O_OPT	!+theta+!	
	p4: _____; O_OPT	!+Z component+!	
+R_ORIENT_*i*_*j*+	p1: orientation; I	!+orientation diff+!	
	p2: real; O	cosine($x_i \rightarrow y_j$)	
+SPHER_VECTOR+	p1: _____; I	!+vector+!	
	p2: _____; O_OPT	!+magnitude+!	
	p3: angle; O_OPT	!+theta+!	
	p4: angle; O_OPT	!+phi+!	
+XYZ_VECTOR+	p1: _____; I	!+vector+!	
	p2: _____; O_OPT	!+X component+!	
	p3: _____; O_OPT	!+Y component+!	
	p4: _____; O_OPT	!+Z component+!	

Parameters

+CYL_VECTOR+

p1 velocity, p2 and p4 speed; or,
p1 displacement, p2 and p4 distance; or,
p1 accel-vec, and p2 and p4 accel

+SPHER_VECTOR+

p1 velocity and p2 speed; or,
p1 displacement and p2 distance; or,
p1 accel-vec and p2 accel

+XYZ_VECTOR+

p1 velocity, and p2, p3, p4 speed; or,
p1 displacement, and p2, p3, p4 distance; or,
p1 accel-vec, and p2, p3, p4 accel

Effects

All programs produce component equivalents from the given vector. No coordinate system frame of reference is assumed by this module; the frame of reference that users employed to initialize the vector will be the one that the components correspond to. In addition:

+ANGLE_ORIENT_*i*+	0° ≤ p2.
+ANGRATE_ORIENT_RATE_*i*+	0°/hour ≤ p2.
+CYL_VECTOR+	p2, p3, and p4 are set to the cylindrical coordinate components of the vector given by p1.
+R_ORIENT_*i*_*j*+	0 ≤ p2 ≤ 1.
+SPHER_VECTOR+	p2, p3, and p4 are set to the spherical coordinate components of the vector given by p1.
+XYZ_VECTOR+	p2, p3, and p4 are set to the x, y, and z Cartesian components (respectively) of the vector given by p1.

2.5.6. Operations Converting to Vector Types

Program	*Parameters*	*Description*	*Undesired events*
+ORIENT_ANGLE+	p1: angle; I	first !+euler angle+!	
	p2: angle; I	second !+euler angle+!	
	p3: angle; I	third !+euler angle+!	
	p4: orientation; O	!+orientation diff+!	
+ORIENT_R+	p1: real; I	cosine($x_1 \rightarrow y_1$)	
	p2: real; I	cosine($x_1 \rightarrow y_2$)	
	p3: real; I	cosine($x_1 \rightarrow y_3$)	
	p4: real; I	cosine($x_2 \rightarrow y_1$)	
	p5: real; I	cosine($x_2 \rightarrow y_2$)	
	p6: real; I	cosine($x_2 \rightarrow y_3$)	
	p7: real; I	cosine($x_3 \rightarrow y_1$)	
	p8: real; I	cosine($x_3 \rightarrow y_2$)	
	p9: real; I	cosine($x_3 \rightarrow y_3$)	
	p10: orientation; O	!+orientation diff+!	
+ORIENT_RATE_ANGRATE+	p1: angrate; I	1st !+euler angle+! change rate	
	p2: angrate; I	2nd !+euler angle+! change rate	
	p3: angrate; I	3rd !+euler angle+! change rate	
	p4: orient_rate; O	!+orient rate diff+!	
+VECTOR_CYL+	p1: _____; I	!+radius+!	
	p2: angle; I	!+theta+!	
	p3: _____; I	!+Z component+!	
	p4: _____; O	!+vector+!	

+VECTOR_SPHER+	p1: _____; I	!+magnitude+!	
	p2: angle; I	!+theta+!	
	p3: angle; I	!+phi+!	
	p4: _____; O	!+vector+!	
+VECTOR_XYZ+	p1: _____; I	!+X component+!	
	p2: _____; I	!+Y component+!	
	p3: _____; I	!+Z component+!	
	p4: _____; O	!+vector+!	
			%range exceeded%

Parameters

+VECTOR_CYL+

p1 and p3 speed, p4 velocity; or,
p1 and p3 distance, p4 displacement; or,
p1 and p3 accel, p4 accel-vec

+VECTOR_SPHER+

p1 speed and p4 velocity; or,
p1 distance and p4 displacement; or,
p1 accel and p4 accel-vec

+VECTOR_XYZ+

p1, p2, p3 speed and p4 velocity; or,
p1, p2, p3 distance and p4 displacement; or,
p1, p2, p3 accel and p4 accel-vec

Effects

+VECTOR_CYL+	p4 = the vector equivalent of p1, p2, and p3, assuming a cylindrical coordinate system.
+VECTOR_SPHER+	p4 = the vector equivalent of p1, p2, and p3, assuming a spherical coordinate system.
+VECTOR_XYZ+	p4 = the vector equivalent of p1, p2, and p3, assuming a Cartesian coordinate system.

2.5.7. Operations Converting from One Vector Type to Another

Program	*Parameters*	*Description*	*Undesired events*
			%range exceeded%
+CHANGE_COOR+	p1: ______; I	vector in 1st frame	
	p2: orientation; I	!+orientation diff+!	
	p3: ______; O	vector in 2nd frame	
+INVERT_ORIENT+	p1: orientation; I		
	p2: orientation; O		
+ORIENT_RATE_VEL+	p1: distance_inv; I	!!conv matrix!!	

	p2: velocity; I p3: orient_rate; O
+VEL_ORIENT_RATE+	p1: distance_inv; I !!conv matrix!! p2: orient_rate; I p3: velocity; O

Parameters

+CHANGE_COOR+	p1 and p3 must belong the same vector typeclass.
Other	A !!conv matrix!! is given as nine separate parameters, whose order is defined in the definition of !!conv matrix!!.

Effects

+CHANGE_COOR+	Let p2 be the !+orientation diff+! from reference frame 1 to reference frame 2. If p1 is a vector in coordinate frame 1, then p3 is vector p1 in coordinate frame 2.
+INVERT_ORIENT+	If p1 is the orientation difference from coordinate frame 1 to coordinate frame 2, then p2 is the orientation difference from coordinate frame 2 to coordinate frame 1.
+ORIENT_RATE_VEL+	p3 = the orient_rate equivalent to p2 with !!conv matrix!! p1 in a Cartesian coordinate system.
+VEL_ORIENT_RATE+	p3 = the velocity equivalent to p2 with !!conv matrix!! p1 in a Cartesian coordinate system.

3. Local Type Definitions

Local types used in the description of programs specified in EC.DATA are defined in section EC.DATA.3, except for the following:

attribute

For a !!scalar type!!, the same as a real attribute (see EC.DATA.3) except that !!EC.range!! and !!EC.resolution!! must be given by entities of that same typeclass.

For types accel-vec, displacement, orient_rate, and velocity:

(*keyword ranres1 ranres2 ranres3* EXACT_REP)

where

ranres ::= lower-bound upper-bound !!EC.resolution!!

The fourth element is optional, and has the same effect as in EC. *keyword* is either SPHER, CYL, XYZ, or blank.
If XYZ or blank, then each *ranres* gives the range and resolution of the X, Y, and Z scalar Cartesian components of the vector.
If SPHER, then *ranres2* and *ranres3* are angle attributes and give the range and resolution of the θ and ϕ components, respectively, of the vector; and *ranres1* is a scalar giving the range and resolution of the radius component of the vector.
If CYL, then *ranres1* and *ranres2* are as for SPHER, and *ranres3* is as for XYZ.

typeclass

Simple enumerated: accel-vec, displacement, orientation, orient_rate, velocity, or

	the lower-case name of any !!scalar type!!.
version	Simple enumerated; a version name applicable to the given typeclass. The enumeration of version names (and characteristics of each) are listed in Appendix E.

4. Dictionary

Dictionary terms used to describe programs specified in EC.DATA are defined in section EC.DATA.4. The following terms are introduced by this module:

!!basic scalar type!! — One of: ACCEL, ACCELRATE, ANGLE, ANGRATE, DENSITY, DISTANCE, PRESSURE, SPEED, TIMEINT.

!!basic scalar unit!! — The upper-case abbreviation of a unit of measurement associated with a !!basic scalar type!!. These are enumerated in Table AT.NUM.a.

!!conv matrix!! — The matrix of inverse distances representing radii of curvature to convert from angular rates about three Cartesian axes into velocity. Velocity components in the directions of the axes across the top of the matrix are equivalent to instantaneous angular rates around the axes down the side of the matrix, with the inverse of the radius of curvature given by the matrix. Note that if any column is all zero, components of velocity in that axis will be used/generated in the conversion. A zero radius may not be represented in this matrix.

	Axis 1	**Axis 2**	**Axis 3**
Axis 1	p1	p4	p7
Axis 2	p2	p5	p8
Axis 3	p3	p6	p9

!+euler angle+! — The euler angles represent the rotation of a set (x_1,x_2,x_3) of orthogonal axes to obtain the set of axes (y_1,y_2,y_3). The rotation is successive; the set is first rotated about the x_3 axis; then about the new x_1 axis; finally, the set is rotated about the new x_3 axis. All rotations are positive counterclockwise looking from the positive axis towards the negative. The results of the rotations map the x_1 axis parallel to the y_1 axis, the x_2 axis parallel to the y_2 axis, and the x_3 axis parallel to the y_3 axis.

!+magnitude+! — The scalar magnitude of the vector; the distance between the end of the vector and the origin of the coordinate system.

!+orientation diff+! — The orientation of the (y_1,y_2,y_3) axes in the (x_1,x_2,x_3) coordinate system.

!+orient rate diff+! — The rate of change per unit time of !+orientation diff+!.

!!power type!! — A !!power type!! of a !!basic scalar type!! is the !!basic scalar type!! raised to an integer power *n*. *n* is called the !!power type index!! of the !!power type!!. In this module, *n*={-1,2,-2,3,-3}. The name of a !!power type!! is formed by appending the name of the !!basic scalar type!! with _INV, _SQ, _INV_SQ, _CU, or _INV_CU.

!!power type index!! — See !!power type!!. A real is considered to have !!power type index!! of 0. A !!basic scalar type!! has !!power type index!! of 1.

!+phi+! — In spherical coordinates, the angle between the vector and positive z axis; sign of

	the angle is the sign of the vector's z component.
!+radius+!	The distance from the origin to the end of the projection into the x-y plane of the vector.
!!scalar type!!	Any !!basic scalar type!! (except for TIMEINT), or a !!power type!! of any !!basic scalar type!! (including TIMEINT).
!!scalar unit!!	The unit of measurement associated with a !!scalar type!!. Defined in Table AT.NUM.a.
!+theta+!	The angle from the positive y axis to the projection of the vector into the x-y plane; the angle is measured clockwise as seen looking into the x-y plane in the negative z direction.
!+vector+!	The vector type class equivalent of the given components.
!+X component+!	The x component of the given vector.
!+Y component+!	The y component of the given vector.
!+Z component+!	The z component of the given vector.

5. Undesired Event Dictionary

Undesired events described in programs specified in EC.DATA.2 are defined in section EC.DATA.5. In addition:

%illegal sin or cos%	User has supplied a sine or cosine with magnitude greater than 1.

6. System Generation Parameters

#max ADT ascon#	Type: real. The maximum allowable magnitude for an ADT ascon or literal (real equivalent)
#max ADT loadcon#	Type: real. The maximum allowable magnitude for an ADT loadcon (real equivalent)
#max ADT ranres ratio#	Type: real. The maximum allowable magnitude of the ratio of a type's !!range!! to its !!resolution!!
#max ADT range#	Type: real. The maximum allowable magnitude for the absv(upper bound - lower bound) for an ADT type (real equivalent)
#min ADT resolution#	Type: real. The minimum allowable resolution for an ADT entity (real equivalent)

CHAPTER 2

AT.STE: STATE TRANSITION EVENT TYPE CLASS

1. Introduction

1.1. Overview

This module implements State Transition Event (STE) types. Users may define specific types of this type class, and declare entities of those types.

Each STE type is a class of equivalent finite state machines. The value of an STE variable is its state. Each STE type is characterized by a set of states, named subsets of that set, named relations on that set, and state transition events. It is intended that this module be used only when the number of states is small enough that the attributes of the type can be described by enumeration.

The relations may be used either to specify conditions relating two entities of the same STE type or to describe state transitions.

This module provides facilities that allow a process to await specified conditions and transitions. Awaiting a state transition event suspends the process until the event occurs. Awaiting a condition suspends the process until the condition holds.

1.2. Literals

Literals are state names and must begin and end with the character "$".

2. Interface Overview

2.1. Declaration of Specific Types

STE types are declared using the form of ++DCL_TYPE++ (see EC.DATA.2.1). The forms of the parameters for STE type declarations are given below.

Parameter	Type	Comments
p1	name	any unused name
p2	typeclass	"STE"
p3	attributes	see AT.STE.3
p4	version	omitted

Effects

Defines an STE type with name p1 and with attributes p3. Entities and arrays of type p1 may now be declared. For each relation, set, and conversion attribute provided, there is a set of run-time access programs that may be defined. These access programs are defined by declaring the access program name in the ‘program_list’ of the attribute declaration (see AT.STE.3). The set of legal program names follow the conventions given in the table below. In all cases, *type* is replaced by the name of the specific type given in the declaration.

Attribute	Legal access program names	Comments
conversion	+*type_conversion*+	where *conversion* is replaced by the first or second ID in the conversion attribute definition.
set	+*type_set*+ +AWAIT.T/F.*type_set*+ +AWAIT.T/F.g.*type_set*+ +AWAIT@T/F.*type_set*+ +AWAIT@T/F.g.*type_set*+	where *set* is replaced by the ID in the set attribute definition.
relation	+IS_*type_relation*+ +*type_relation*+ +AWAIT@*type_relation*+ +AWAIT@.g.*type_relation*+ +AWAIT@=*type_relation*+ +AWAIT@=.g.*type_relation*+ +AWAIT.T/F.*type_relation*+ +AWAIT.T/F.g.*type_relation*+	where *relation* is replaced by the ID in the relation attribute definition.

The syntax and semantics of these programs are specified in section AT.STE.2.5.1.

The following additional undesired events can occur in the declaration of STE types:

%%duplicate set member%%
%%duplicate program name%%
%%malformed attribute%%
%%malformed program name%%
%%missing state attribute%%
%%too many conversion attributes%%
%%too many relation attributes%%
%%too many set attributes%%
%%too many state attributes%%
%%too many states%%
%%type inconsistency%%
%%undeclared spectype%%
%%unknown state%%

2.2. Declaration of Variables and Constants

STE variables and constants are defined using the form of ++DCL_ENTITY++ (see EC.DATA.2.2).

2.3. Declaration of Arrays

Arrays of STE type entities are defined using the form of ++DCL_ARRAY++ (see EC.DATA.2.3).

2.4. Operand Descriptions

See description in EC.DATA.2.4. The following additional undesired event can occur when operands are specified: %%inappropriate parameters%%

2.5. Access Programs

The following additional undesired event can occur when an STE access program is invoked: %%unknown STE program%%

2.5.1. Operations on STE Entities

Program	*Parameters*	*Description*	*Undesired events*
+*type_set*+	p1: STEtype; I	entity to test	
	p2: boolean; O	destination	
			None
+IS_*type_relation*+	p1: STEtype; I	pair to test	
	p2: STEtype; I		
	p3: boolean; O	destination	
	p4: (parms); I	n-tuple	
			%%unequal lists%%
+*type_relation*+	p1: STEtype; IO	domain elements	
	p2: (parms); I	n-tuple	
			%%unequal lists%%
			%out of domain%
			%out of domain%
+*type_conversion*+	p1: STEtype; I	value to convert	
	p2: convtype; O	destination	
+*type_conversion_2*+	p1: convtype; I	value to convert	
	p2: STEtype; O	destination	

Effects

+*type_set*+ p2 := (p1 is an element of the set attribute whose identifier is *set*)

+IS_*type_relation*+ p3 := ((p4,p1),p2) is an element of the relation attribute whose identifier is *relation*. If p4 is an empty list then, p3 := (p1,p2) is an element of *relation*.

+*type_relation*+ p1 changes state to s2 such that s1 is the value of p1 before the call, v1,...,vn is the value of p2 (if given), and the ordered pair (((v1,...vn),s1),s2) (or (s1,s2) where p2 is an empty list) is an element of the relation attribute whose identifier is *relation*. If s2 has the value NC (no change), the function has no effect. Where there is more than one ordered pair with the same first element, any may be chosen.

+*type_conversion*+ p2 := x such that (p1,x) is an element of the conversion relation given by *type_conversion* where *type_conversion* is 'ID1' of a 'conversion_defn'.

+*type_conversion_2*+ p2 := x such that (p1,x) is an element of the inverse relation *type_conversion_2* where *type_conversion_2* is given as 'ID2' of a 'conversion_defn'.

2.5.2. Await Operations

In this section ''AWAIT.T/F.*string*'' stands for two names, ''AWAIT.T.*string*'' and ''AWAIT.F.*string*''. Similarly, in the effects section, alternate phrasing for each member of the pair is separated by ''/''.

Some of the await programs take a list of parameters, written "(parms)", as a parameter. Any variables in such a list are evaluated exactly once at the time the await operation begins execution. The value of the variable at that time is the value used by the await operation even if the value of the variable changes while the process executing the await is waiting.

Program	*Parameters*	*Description*	*Undesired events*
			%%No STE constant%% %%No STE variable%%
+AWAIT@*type_relation*+	p1: STEtype; I p2: (parms); I	variable to watch n-tuple	
+AWAIT@.g.*type_relation*+	p1: STEtype; I p2: (parms); I p3: STEtype; O	variable to watch n-tuple post transition state	
+AWAIT@=*type_relation*+	p1: STEtype; I p2: (parms); I	variable to watch n-tuple	
+AWAIT@=.g.*type_relation*+	p1: STEtype; I p2: (parms); I p3: STEtype; O	variable to watch n-tuple post transition state	
+AWAIT@T/F.*type_set*+	p1: STEtype; I	candidate member	
+AWAIT@T/F.g.*type_set*+	p1: STEtype; I p2: STEtype; O	candidate member post transition state	
+AWAIT.T/F.*type_set*+	p1: STEtype; I	candidate member	
+AWAIT.T/F.g.*type_set*+	p1: STEtype; I p2: STEtype; O	candidate member post transition state	
+AWAIT.T/F.*type_relation*+	p1: STEtype; I p2: STEtype; I p3: (parms); I	candidate pair n-tuple	
+AWAIT.T/F.g.*type_relation*+	p1: STEtype; I p2: STEtype; I p3: (parms); I p4: STEtype; O	candidate pair n-tuple post transition state	

Effects

+AWAIT@*type_relation*+	Let p2’ be the value of parameter(s) p2. Then, all action in the calling process is suspended until p1 undergoes a state change from s1 to s2 such that the ordered pair ((p2’,s1),s2) (or (s1,s2) if p2 is an empty list) is an element of the relation attribute whose identifier is *relation*.
+AWAIT@.g.*type_relation*+	As for +AWAIT@*type_relation*+ with the addition that p3 := s2.
+AWAIT@=*type_relation*+	If p2 is an empty list, then all action in the calling process is suspended until the access program +*relation*+(p1) is executed. Otherwise, let p2’ be the value of p2. Then all action in the calling process is suspended until the access program +*type_relation*+(p1,p2) is executed with p2=p2’.

+AWAIT@=.g.*type_relation*+	As for +AWAIT@=*type_relation*+ with the addition that p3 := s2 where s2 is the value of p1 immediately following execution of the +*type_relation*+() access program.
+AWAIT@T/F.*type_set*+	Let S be the set of states defined by the attribute *set*. Then all action in the calling process is suspended until p1 undergoes a state change from s1 to s2 such that s1 is not/(is) an element of S and s2 is/(is not) an element of S.
+AWAIT@T/F.g.*type_set*+	As for +AWAIT@T/F*type_set*+ with the addition that p2 := s2.
+AWAIT.T/F.*type_set*+	If the value of p1 is/(is not) a member of the set attribute whose identifier is *set*, the call has no effect. Otherwise, all action in the calling process is suspended until the value of p1 is/(is not) a member of *set*.
+AWAIT.T/F.g.*type_set*+	As for +AWAIT.T/F.*type_set*+ with the addition that if the process is not suspended, p2 := p1; if the process is suspended, p2 := s2 where s2 is the value of p1 after the call that causes the process to become unsuspended.
+AWAIT.T/F.*type_relation*+	Let p3' be the value of parameter(s) p3. Then, all action in the calling process is suspended until the ordered pair of values given by ((p3',p1),p2) (or (p1,p2) if p3 is an empty list) is/(is not) a member of the relation attribute whose identifier is *relation*. If the ordered pair of values is/(is not) already a member of *relation*, the call has no effect.
+AWAIT.T/F.g.*type_relation*+	As for +AWAIT.T/F.*type_relation*+ with folowing addition. Let p' be whichever of p2, p3 is non constant. Then, if the process is not suspended p4 := p'. If the process is suspended p4 := s2 where s2 is the value of p' after the call that causes the process to become unsuspended.

3. Local Type Definitions

Local types that are used in the description of programs specified in EC.DATA and are not described here are defined in section EC.DATA.3. In the following definitions, where the syntax is given in modified BNF, alphabetic strings in italics (e.g. *state*) are terminal symbols. Ellipses (e.g. *state* ... *state*) denote a list of two or more elements separated by one or more blanks.

attribute — the attributes of an STE type characterize the type. The syntax of an STE type attribute is:

```
::=    (state state_list)
 |     (relation (relation_defn))
 |     (set (set_defn))
 |     (conversion (conversion_defn))
```

An STE type declaration must have exactly one *state* attribute. There may be any number of the remaining attributes.

attributes — ::= (attribute_list)

attribute_list — ::= attribute | attribute attribute_list

convtype — An entity of the specific type specified in the conversion_defn.

conversion_defn — A mapping between a subset of the states of an STE type and a set of literals of another type. The syntax is:

```
::=    ID1 (program_list) type_ID (conv_pairs)
```

```
| ID1 ID2 (program_list) type_ID (conv_pairs)

conv_pairs ::= (state_list lit_con)
            | (state_list lit_con) conv_pairs
```

The ‘type_ID’ must be the identifier of a specific type and ‘lit_con’ must be a literal or constant of that type. ‘ID1’ gives the identifier for the relation given by the set of ‘conv_parirs’. ‘ID2’, if given, denotes the inverse relation. The conversion program specified by the relation, its inverse or both may be requested by listing their names in the ‘program_list’. If the program corresponding to relation ‘ID2’ is requested and ‘ID1’ is not a one-to-one relation, then the conversion program may return any element of the corresponding ‘state_list’.

ID	an identifier associated with an STE attribute
parms	a list of entities. The number of entities and their types must match those of the n-tuple "P" in the corresponding relation_defn.
program_list	::= access_program | access_program ... access_program where "access_program" is the name of one of the access programs constructed according to the templates in AT.STE.2.5.
relation_defn	a set of ordered pairs of the form ((P D) R) where (P D) is an element of the domain of the relation being defined and R is an element of the range. D and R are m-tuples. The elements of D must be elements of the state set. Elements of R must either be elements of the state set or the special symbol NC. P may be an n-tuple of literals of any type. P may be omitted, in which case the ordered pairs are written (D R). The syntax is:

```
::= ID (program_list) (ordered_pairs)

ordered_pairs ::= (domain state_list) | (domain state_list) ... (domain state_list)

domain ::= state_list |  (literals state_list)

literals ::= literal | (literal ... literal)
```

set_defn	a subset of the set of states of the STE type being defined or a subset of the power set of the set of states. The syntax is,

```
::=     ID (program_list) state_list
|       ID (program_list) (state_list ... state_list)
```

state	an STE literal (i.e. state name).
state_list	::= state | (state ... state)
STEtype	an entity of the STE type for which the operation is defined or a list of such entities enclosed in parentheses and separated by one or more blanks.

4. Pre-defined Relations

Certain relations are automatically defined on every STE type. Programs associated with these relations may be used after giving a declaration as described below.

Let S be the state set of an STE type and let s1, s2 be elements of S. Then the following relations are defined on the state set.

*type*_EQ	The set of ordered pairs (s1 s1).

*type*_SET	The set of ordered pairs ((s2 s1) s2).
*type*_CHANGED	The set of ordered pairs ((s1 s1) s2)

Access programs associated with a pre-defined relation are declared by giving a 'relation_defn' for the relation without the corresponding list of ordered pairs. Otherwise, the syntax and semantics of the declaration are the same as those for the corresponding 'relation_defn'. The name of the pre-defined relation must be prefixed with the name of the STE type being defined. The format is,

relation (*type*_relation_name (program_list) ())

for instance,

relation (*type*_EQ (+IS_EQ+ +AWAIT.T.EQ+) ())

declares the access programs +IS_EQ+ and +AWAIT.T.EQ+ associated with the EQ relation for the STE type "type".

5. Dictionary

None.

6. Undesired Event Dictionary

%%duplicate set member%%
: a user program has specified a set with duplicate elements.

%%duplicate program name%%
: a user has tried to define an access program name that has already been defined.

%%inappropriate parameters%%
: a user program has called an access program with parameters that do not correspond in specific type to the declaration.

%%malformed attribute%%
: the user has supplied an attribute declaration with a syntax inconsistent with that specified in AT.STE.3.

%%malformed program name%%
: the user has declared a program name in the program_list of an attribute definition that is inconsistent with the set of legal program names for the attribute.

%%missing state attribute%%
: a user program has declared an STE type with no "state" attribute.

%%No STE constant%%
: an access program of the form +AWAIT.T/F.*relation*+ or +AWAIT.T/F.g.*relation*+ is called and neither of p1 and p2 is an STE literal or constant.

%%No STE variable%%
: an access program of the form +AWAIT.T/F.*relation*+ or +AWAIT.T/F.g.*relation*+ is called and neither of p1 and p2 is an STE variable.

%out of domain%
: a user program has supplied an input parameter that is not in the domain of the relation, conversion relation, or inverse conversion relation.

%%too many conversion attributes%%
: a user program has declared an STE type with more than the allowable number of conversion attributes, as given by a sysgen parameter.

%%too many relation attributes%%
: a user program has declared an STE type with more than the allowable number of relation attributes, as given by a sysgen parameter.

%%too many set attributes%%
: a user program has declared an STE type with more than the allowable number of set attributes, as given by a sysgen parameter.

%%too many state attributes%%
: a user program has declared an STE type with more than one "state" attribute.

%%too many states%%
: a user program has declared an STE type with too many states, as given by a sysgen parameter.

%%type inconsistency%%
: a user program has supplied a conversion value not of the specified type.

%%undeclared spectype%%
: a user program has specified a specific type that has not been defined.

%%unequal lists%%
: a user program has supplied a list parameter that does not match the specified length.

%%unknown state%%
: a user program has provided a state value that has not been declared in the "state" attribute.

%%unknown STE program%%
: a user program has specified an STE program that has not been defined.

7. System Generation Parameters

#max conversion attributes# (integer)
: The maximum number of conversion attributes allowed in a type attribute.

#max relation attributes# (integer)
: The maximum number of relation attributes allowed in a type attribute.

#max set attributes# (integer)
: The maximum number of set attributes allowed in a type attribute.

#max state list size# (integer)
: The maximum number of states allowed in a state attribute of a type.

APPENDIX A

Design Issues

1. AT.NUM Design Issues

(1) While AT and EC designers are critically concerned with which facilities are machine-dependent and which are not, we felt that users of the resulting modules would not be as concerned. Therefore, we tried to design this module so that users could regard it as an extension to the Extended Computer; the AT and EC in conjunction provide the basic programming environment for the rest of the system. We tried to make the design as parallel to EC.DATA as possible, in order to present in effect a single kind of interface to programmers interested in manipulating data of any type.

(2) We originally thought that registers for abstract data types would not be useful. If they turned out to be desirable, however, the facility could be a straightforward extension, based on the EC architecture. Later we realized that the same thing that made them useful in the EC (namely, the storage savings that the allowed) made them useful in AT.

(3) Any design issue in EC.DATA concerning facilities or designs copied by this module naturally also applies to this module.

(4) We used to disallow giving a sine/cosine pair to specify an angle literal. This was (according to an earlier design issue) "to keep the interface simple and consistent." However, the interface of that version also failed to allow any way to specify vector literals, and once we added the necessary syntax for that, allowing sine/cosine angle literals was trivial and clearly consistent.

(5) We used to have an abstract data type called "mach", using it to represent a speed relative to the speed of sound. But it produced so many hard questions (e.g., when you divide a speed by a speed, do you get a real or a mach? How can you check at compile-time? If the speed you compare with is the speed of sound at some other altitude/pressure/temperature, is the result a mach or not?) that we decided it wasn't really an abstract data type like the others.

(6) One of the aircraft devices takes an angle that is represented in "octant/magnitude" form. The octant is an integer from 0 to 7; the magnitude is an angle from 0 to 45 degrees. The angle they define may be thought of as (45° × octant) + magnitude. (The magnitude happens to be represented by a complex function of the tangent or cotangent of the angle, but this is irrelevant.) Should ADT provide a version of angle (including conversion routines) that corresponds to this representation? We decided not, because the module responsible for driving the device can obtain the necessary representation by performing operations on the angle represented in degrees, or as a sine/cosine pair. (This assumes the presence of a tangent and cotangent routine somewhere in the system, a reasonable assumption.) A weaker reason is that such a representation is only expected to be used in one place in the whole OFP, and this goes against the rule of having ADT implement "generally useful" data types and data type operations.

(7) We added the degree/minute/second angle conversion routines because certain panel displays need to display angles (such as latitudes) in that fashion, rather than simply as degrees and fractions of degrees. It might be more efficient to store such angles in three parts, rather than computing the minutes and seconds from fractions of degrees. That decision clearly rests in this module, and so the facilities were added so that users need not do it themselves. We also added a new angle version consistent with this possible representation.

(8) We state as an assumption that it is adequate to maintain angles with a 360-degree range. Although this assumption is likely to prove correct, it is actually more expensive for this module to limit all angles to between 0 and 360 degrees, or between -180 and +180 degrees. For one thing, it would mean that every time an angle was computed, we would have to perform a modulo operation, a very expensive proposition. Until we realized this, the interface promised that all angles would be limited in that fashion. We thought about dropping that promise and providing three kinds of angle-to-real conversion programs:

one that returned a value from 0-360 degrees, one that returned a value from -180 to +180 degrees, and one that returned a value not limited to one circle's worth. However, we deemed this inappropriate because it implies that -90 degrees is the same angle as 270 degrees, and this conflicts with classic geometry. Further, it would not allow us to hide whether or not we limited the angles internally. Let *x* be set to 300 degrees + 200 degrees. Then +GT+(*x*, 150 degrees) has a different answer depending on whether we do or do not limit internally. We decided that it should be up to the user to tell us when to perform the limiting, and it should be incumbent upon him to pay the corresponding price in efficiency.

(9) We need inverse time intervals in our application, as well as timeint^2, but decided that the representation of such types was not part of the EC's secret. Hence, although timeint is a type provided by EC, its derivatives are provided by this module. We believe that we can efficiently implement the operations on those types without knowledge of our host computer; if that turns out not to be the case, we will move the operations into the EC.

(10) The present OFP computes arctangent ("angle whose tangent is equal to...") of certain numbers. However, it appears as though the input to that computation is always a sine/cosine pair, as opposed to a single real representing the tangent. This prevents some loss of precision. That means that the AT conversion program to produce an angle from a sine/cosine pair will be useful, and we have designed our interface accordingly.

(11) It is important to specify that the +ABSV+ returns the Euclidean norm of a vector because (a) there are several instances where our programmers need to compute $\text{SQRT}(a^2+b^2)$, and they couldn't use this program unless they were sure it gave that result; and (b) there are literally an infinite number of ways to compute the norm of a vector, each giving a different answer, so it is important to specify which one we use.

(12) After we decided to allow angles of any magnitude, we realized that our design had an inconsistency. To assign a value to an angle larger than 360°, a sine/cosine pair could not be used. There are an infinite number of angles with a given sine/cosine pair, each differing by one circle, and there is no way to tell which one is meant. We considered four alternatives:

(a) disallow the assignment of angle values using a sine/cosine notation; add undesired events forbidding it. We rejected this because it violates our notion of versions; namely, all versions of a typeclass are semantically equivalent. This would have made the sine/cosine version different.

(b) remove sine/cosine angles from our list of versions, recognizing that sines and cosines are simply real numbers. This would be similar to our decision removing mach from the list of ADT types. We rejected this because sine and cosine seem a perfectly reasonable way to represent angles.

(c) change what we mean by versions. We rejected this because it would change a basic principle of our design for not a very good reason.

(d) add a third parameter to sine/cosine pairs, specifying the number of 360° circles to add to the specified angle. This seemed the most mathematically sound, and most consistent with our design. We made the parameter to the appropriate conversion program optional so that existing code need not change; the default is zero.

(13) Parnas has suggested that some of the problems with angle ranges discussed above have arisen because we are using angles "to represent two completely different kinds of information. One may be thought of as the angle coordinate of a polar coordinate system, the other has to do with angular displacement or movement. When we are thinking of a polar coordinate system, -90° and +270° are indeed equivalent. When we are thinking of the amount of rotation necessary to remove a screw or open a faucet, they are not." He suggested that a better design might be to have had two different angle types, one for each of the two applications mentioned above.

(14) Our conversion programs between vectors and spherical or cylindrical coordinates used to define the theta coordinate as the angle from the positive x axis measured counterclockwise; this matched the classical math text definitions. However, it turns out that it is not the most useful for an avionics application dealing with angles like true headings. One of the improvements that we had hoped to make in our version of the OFP was to eliminate the many different sets of coordinate systems found in the NWC program. The old theta definition, when used to track a heading or a wind direction, was incompatible with our east-north-up right-handed reference frame used most widely throughout our software.

(15) The addition of the accel-rate typeclass came after values with units of ft/sec^3 starting turning up with some regularity in the IMS equations. It was a ten-minute change to the specification, given all the generic infrastructure already present for basic scalar types.

(16) We probably should have a way of letting a user specify a *set* of versions for a spectype, and allow the implementation to choose any member of that set. Programmers choose versions based on the ways in which the entities will be used in their programs, and we have seen several cases where manipulation would be just as efficient in one version as another. For instance, there is little difference in the efficiency of manipulating angles in degrees as opposed to angles in circles. Since the programmer has to pick one, the choice might be arbitrary; suppose the choice is degrees, and suppose that one of the entities is a parameter to the program being written. If the caller of that program has all of its angles in circles, then a conversion will have to inserted in the implementation of the program call. But if the implementor had the choice circles or degrees, circles could have been chosen in the first place.

2. AT.STE Design Issues

(1) Originally, the AT module included only the synchronization operations +SIGNAL+ for signalling the occurrence of events and +AWAIT+ for allowing processes to wait for the occurrence of events. These operations were chosen because they support the notion of events in the OFP specifications.

(2) These +AWAIT+ operations originally were part of the EC interface. They were moved to the AT module because (a) they can be constructed using the +UP+, +DOWN+ and +PASS+ operations provided by the EC, (b) the EC synchronization operations would not be made simpler by using +AWAIT+ and +SIGNAL+, (c) we could think of useful systems that did not need +AWAIT+ and +SIGNAL+ and (d) it made the EC interface simpler.

(3) It was noted that, in some cases, processes needed to wait if the system was not in a particular state but did not need to wait if the the system was already in the desired state. (Here and in the following paragraphs, "state of the system" will be used to mean a particular system state or a class of such states.) Simple event variables could not be used in this case since the event associated with the system entering a state would not be signaled until the state had been left and entered again. To solve this problem, we added an additional event variable to the interface called an "event boolean", and an additional await operator. Processes could wait on the event of a state change in an event boolean using the +AWAIT+ operator. The new operator, +AWAITC+ (await on condition), caused the process to wait only if the event boolean was not in the specified state. If the event boolean was already in the specified state, the calling process continued without interruption. This allowed processes to respond to an event that occurred while the process was active. This facility was later superceded (see 4.).

(4) The specifications require that processes be able to wait on the occurrence of complex events, for instance, the disjunction of two or more events or the occurrence of an event while the system in in a particular state (i.e. @T(event) WHEN (condition AND condition AND ...)). Complex synchronization conditions could not be implemented directly with event booleans so we attempted to provide a syntax for expressing complex events and a synchronization mechanism for interpreting that syntax. Users of the synchronization module would express the complex event for which they wanted to wait in the syntax provided by this module. The module would interpret the requests, translate them into more primitive synchronization operators and signal the appropriate events. This scheme proved inadequate for several reasons. The synchronization module proved to be large and complicated. The interpretation mechanism was complex and difficult to implement. The proposed mechanism still did not solve all of the problems associated with waiting for complex events:

(a) if user processes were awaiting a disjunction of events, they could not distinguish which event had been signalled. Processes could not perform conditional actions based on the awakening event without checking the values of extra variables. Even with very short deadlines, these values could change before they were checked.

(b) processes waiting on events with WHEN clauses frequently needed to know if the conditions expressed in the WHEN clause still held by the time they began running. These processes would have to recheck the condition values after they began running.

(c) since the events that processes awaited did not necessarily reflect the state of the system by the time they began running, the order in which these events had occurred could not necessarily be determined. Processes that needed to run in a particular order depending on the order of events were not able to do so.

(d) event variables and event booleans, these problems could be solved only by providing great numbers of small short deadline processes that recorded the occurrences of events in local variables.

(5) STE variables were proposed as a replacement for event variables and event booleans. Users can define event variables with more than two states and can wait on state change events or transitions between states. This mechanism allows us to solve those problems listed in (3) where the class of states is small enough to describe by enumeration and the transition time between states is large enough that the changes can be recorded by the event detection mechanism. For instance,

(a) event booleans are unnecessary since they can be implemented with STEs.

(b) STEs provide a single mechanism for signalling transitions in variables with more than two discrete states.

(c) a syntax for expressing awaits on complex events is unnecessary. A state of an STE variable can be associated with the occurrence of a complex event and the user process need only wait on the transition to that state.

(d) user processes can wait on a disjunction of events and determine which event caused the signal by examining the value of an STE variable.

(e) STEs can record the "history" of events in their transitions. In those cases where the history can be represented by an enumerated set of states and transitions, STEs allow the user processes to respond to events in the order of their occurrence and to events that have occurred while they were running. Additional processes need not be created to perform these tasks.

(f) processes can signal different events to different users by causing a transition in a single STE variable.

(6) Originally, the STE module provided comparison operators for STE values, set membership operators and conversion operators. We found that users of the module frequently only needed a small subset of the operations provided. By allowing the creator of the STE type to define the operations needed, the required subset of operations can be selected without incurring overhead for unneeded ones.

(7) Originally the STE module provided for one explicit ordering on a given STE domain. The comparison operators provided referred to that ordering. We decided that such an ordering was unnecessary as it can be implemented as a special case of a relation.

(8) Originally the STE module did not provide AWAIT programs that return a value. We expect it to be fairly common for programmers to AWAIT a transition then ask for the value of the STE variable concerned. Since that value could change between the time of the transition and the time the variable is read, exclusion relations would have to be used to ensure that the correct value is read. We decided it would be simpler and potentially more efficient for the STE modules to provide such values where they are wanted.

(9) The access programs +AWAIT@*type_set*+ and +AWAIT@.g.*type_set*+ are redundant since the same programs can be provided by defining an equivalent relation and using the await program for that relation. These programs are provided on the interface because the relation definitions rapidly become unwieldy as the cardinality of the state set increases.

(10) An AWAIT program for transition out of a set was not originally provided since the same thing can be accomplished by defining the inverse set. However, it makes the STE declarations less clear and much longer than necessary. To simplify the defintions and their use, the AWAIT programs for sets were changed to allow processes to wait for transition both into and out of a set of values.

APPENDIX B

Implementation Notes

3. AT.NUM notes

(1) There are occurrences in NWC-2 (location 0A62, as called by 1E35 - 1E44) of conversions of a single precision angle to a double precision sine/cosine pair. The user of this module would probably accomplish that by providing an entity declared to be of an angle version other than AN4 that would fit into a single word as input to +R_ANGLE_COS+ or +R_ANGLE_SIN+ with the output parameter having a range/resolution ratio requiring a double word. The implementation should be prepared to handle this.

(2) An algorithm for computing arc tangents may be found in [WPN], pages 86-87.

(3) The implementation should be sensitive to vector spectypes specified with a third element having a range of zero. These vectors can be represented using two EC.real entities rather than the usual three.

APPENDIX C

Assumptions Lists

1. AT.NUM Assumptions

1.1. Basic Assumptions

All basic assumptions in Appendix 3 of [EC] apply, except those that explicitly mention bitstrings, registers, or time intervals. Where an EC.DATA assumption mentions the Extended Computer or EC, readers of this module should substitute "Application Data Types module" or "AT", respectively. In addition, the following assumptions apply to this module:

(1) For each scalar type class provided, there is a fixed set of units of measurement into which values of that class may be converted. The units for each type class are listed in Table AT.NUM.a. Values of vector type classes may be converted into component scalar values, or a direction/magnitude equivalent.

(2) User programs may not make assumptions about the representation of numeric values. Even though literals are expressed in particular units, there is no implication that the value is stored in those units.

(3) The only operations needed for calculating new numeric values are: addition, multiplication, division, subtraction, absolute value, and complement. In addition, we need to convert scalars to/from reals, and vectors to/from component scalars.

(4) We need arrays of numeric data types in which the attributes of an element can change independently of the attributes of other elements.

(5) The range and resolution of each numeric variable can be determined at the time the system is generated.

(6) Arithmetic operations involving operands of different type classes may or may not have a useful physical meaning. Those that do are those specified as legal combinations in the +MUL+ and +DIV+ programs, and no others.

Further, if a value of any type class is multiplied or divided by a real, the result has the same type class.

(7) Some users need angles limited to a range between 0 and 360 degrees (or the equivalent in other units). Some users need angles limited to a range between -180 and +180 degrees. Some users need angles with an arbitrary range.

1.2. Assumptions about Undesired Events

See Appendix 3 of [EC]. All Undesired Event assumptions there apply, except for those that explicitly mention bitstrings, registers, or time intervals. In addition:

(1) User programs will not provide a real with magnitude greater than 1 to represent a sine or a cosine of an angle.

1. AT.STE

1.1. Basic Assumptions

(1) State transitions may be considered to be instantaneous.

(2) When processes need to wait for particular states to occur, they do not need to proceed while waiting. The states for which the process is waiting can be determined before the process begins to wait.

(3) If a process is awaiting a state change, it need not proceed until the change occurs. The transitions for which the process is waiting can be determined before the process begins to wait.

(4) There is no need for a mechanism that allows a process to be started before the state or transition that has been specified in an AWAIT operation has occured.

1.2. Assumptions about Undesired Events

(1) Users will not supply an argument to a conversion outside of the domain of the defining relation.

APPENDIX D

Unimplemented Features

Not all of the capabilities described in this document have been provided in the current version of the implementation. A few facilities, which are not currently needed by the application program, have not been implemented. An attempt to use an absent facility will result in an undesired event in the development version. The unimplemented features are described below.

1. AT.NUM Unimplemented Features

FEATURE: Some typclasses
UNDESIRED EVENT: %%unimplemented typeclass%%
CURRENT USE: The only !!power type!!s that will be implemented whose !!power type index!! is 3 or -3 will be DISTANCE_CU and DISTANCE_INV_CU.

FEATURE: Entities with attributes that can vary across typeclasses
UNDESIRED EVENT: %%unimplemented binding%%
CURRENT USE: In the ++DCL_TYPE_CLASS++, ++DCL_ENTITY++, and ++DCL_ARRAY++ programs, all elements of the type_list must belong to the same typeclass.

FEATURE: Using variables to specify attributes of a specific type, or of a variable or array with varying attributes
UNDESIRED EVENT: %%unimplemented attribute via variables%%
CURRENT USE: To specify an attribute (as defined in AT.NUM.3), literals or ascons must be used.

FEATURE: Using the EXACT_REP attribute for some resolutions
UNDESIRED EVENT: %%unimplemented EXACT_REP resolution%%
CURRENT USE: For a numeric type to have the EXACT_REP attribute, its resolution must be given a value that is an exact integer power of two in one of the units associated with its typeclass.

In addition, the following Extended Computer unimplemented features apply verbatim (see [EC], Appendix D):
Detection of %uninitialized entity%
!!EC.list!!s of operands to access programs
Checking parameter type when it is given by a pointer
Checking for %constant destination% with pointers
Giving !!user threshold!! as a variable

2. AT.STE Unimplemented Features

None.

APPENDIX E

Version Catalogue

For some numeric data types, the Application Data Types module is capable of providing more than one kind of representation (version). The version has no effect on the outcome of an operation, but some versions allow some operations to be performed more quickly than other versions.

The version catalog lists the provided version names for each AT typeclass. Unless otherwise noted, a typeclass stands for all !!power type!!s of that typeclass. When declaring a specific type, users may request a particular version by using the version names given for that typeclass.

All operations have the property that if all input parameters are of the same version, the operation will be more efficient. If the destination is also of the same version, the operation will be even more efficient. The most efficiency can be achieved by maximizing the number of parameters in an operation that are of the same version.

Typeclass	Version	Properties: good for...
accel	AC1	...converting to/from reals in g's
	AC2	...converting to/from reals in fpss
accel-rate	ACRT1	...converting to/from reals in ft/sec/sec/sec
accel-vec	AC1	...converting to/from reals in g's and Cartesian coords
	AC2	...converting to/from reals in fpss and Cartesian coords
	AC1S	...converting to/from reals in g's and Spherical coords
	AC2S	...converting to/from reals in fpss and Spherical coords
	AC1C	...converting to/from reals in g's and Cylindrical coords
	AC2C	...converting to/from reals in fpss and Cylindrical coords
angle	AN1	...converting to/from reals in degrees from 0 to 360
	AN2	...converting to/from reals in circles from 0 to 1
	AN3	...converting to/from reals in radians from 0 to 2π
	AN4	...converting to/from sine and cosine
	AN5	...converting to/from reals in degrees from -180 to 180
	AN6	...converting to/from reals in circles from -½ to +½
	AN7	...converting to/from reals in radians from -π to +π
	AN8	...converting to/from degs/mins/secs from 0 to 360
	AN9	...converting to/from degs/mins/secs from -180 to +180
angrate	ANRT1	...converting to/from reals in deghour
	ANRT2	...converting to/from reals in radsec
density	DE1	
distance	DS1	...converting to/from reals in ft
	DS2	...converting to/from reals in nmi
displacement	DS1	...converting to/from reals in ft and Cartesian coords
	DS2	...converting to/from reals in nmi and Cartesian coords
	DS1S	...converting to/from reals in ft and Spherical coords
	DS2S	...converting to/from reals in nmi and Spherical coords
	DS1C	...converting to/from reals in ft and Cylindrical coords
	DS2C	...converting to/from reals in nmi and Cylindrical coords
orientation	OR1	
orient_rate	ORRT1	
pressure	PR1	

speed	SP1	...converting to/from reals in fps
	SP2	...converting to/from reals in fpm
	SP3	...converting to/from reals in knots
velocity	SP1	...converting to/from reals in fps and Cartesian coords
	SP2	...converting to/from reals in fpm and Cartesian coords
	SP3	...converting to/from reals in knots and Cartesian coords
	SP1S	...converting to/from reals in fps and Spherical coords
	SP2S	...converting to/from reals in fpm and Spherical coords
	SP3S	...converting to/from reals in knots and Spherical coords
	SP1C	...converting to/from reals in fps and Cylindrical coords
	SP2C	...converting to/from reals in fpm and Cylindrical coords
	SP3C	...converting to/from reals in knots and Cylindrical coords
!!power	TM1	...converting to/from reals in milliseconds
type!!s of	TM2	...converting to/from reals in seconds
timeint	TM3	...converting to/from reals in minutes
	TM4	...converting to/from reals in hours

References

[ACONV] Clements, "Conventions for SCR Application Modules", NRL Technical Memorandum 7590-196:PC:pc, September 1985.

[DI] Parker, Heninger, Parnas, Shore; *Abstract Interface Specification for the A-7E Device Interface Module*; NRLMemorandum Report 4385; November, 1980.

[EC] Britton, Clements, Parnas, Weiss; *Interface Specifications for the SCR (A-7E) Extended Computer Module*; NRL Memorandum Report 4843; May, 1982.

[NCMA] Pizer; *Numerical Computing and Mathematical Analysis*; Science Research Associates, Inc., 1975.

[SO] Clements, Parker, Parnas, Shore; *A Standard Organization for Specifying Abstract Interfaces*; NRL Report 8815; June, 1984.

[WPN] George, *A-7E Aircraft Weapon Delivery Equations*, NWC TM 2926, September 1976.

Acknowledgements

The authors gratefully acknowledge the following people who reviewed this document for consistency, clarity, and correctness:

Naval Weapons Center, China Lake, CA
Ms. Dawn Janney, Ms. Jo Miller

Bell Laboratories, Columbus, OH
Dr. Don Utter

Vought Corporation, Dallas, TX
Mr. Glenn Cooper

Department of Computer Science, University of Victoria, British Columbia
Prof. Michael Levy

In addition, we thank Bruce Labaw and Preston Mullen of NRL, who contributed significantly to the architecture described herein.

INDEX

1. Access programs

2. Local type definitions

3. Dictionary terms

4. Undesired events

5. System generation parameters

Table of Contents